CS2001 Week 10 Practical 160003493 Tutor: Tom Kelsey 21 November 2017

Overview

The requirement of this practical was to produce an implementation of three different sorting algorithms, such that the amount of time the algorithm takes to sort the data can be measured. We then had to write tests to experiment with the performance of the algorithms as various values varied, collect the relevant data, and present and analyse it.

Implementation

I used a Beacon class to represent the beacon objects, which implements Comparable to allow a compareTo method, and the Collections.sort() method to be used. The class has attributes to represent the x and y coordinates, and a double to store the distance from the target. This makes it easier to calculate distances from a single target, but a separate class would probably be required if I wanted to experiment with multiple targets, or a moving target.

The Basic algorithm's findNearestBeacons method is implemented by iterating through each beacon in the ArrayList passed in the constructor. For each beacon, the distance is calculated using Pythagoras' Theorem and stored, then the beacon is inserted into a sorted array, which requires part of the array being copied one index along to make space for the beacon.

The SortOnce algorithm's findNearestBeacons method uses the same method as Basic to calculate and store the distance from the target. To try to accurately compare between algorithms, I use arrays rather than ArrayLists for storage in the algorithm classes where possible. However, in this class I used an ArrayList to store the unsorted beacons, to allow me to use the Collections.sort() method to sort all the beacons at the end. I then copy the required number of nearest beacons to an array which is returned.

The DontSort algorithm's findNearestBeacons method fills the array of nearest beacons while there is space in it, and then iterates through the remaining beacons, checking if they are smaller than the current largest distance in the nearest beacons array, and if so replacing the current beacon. I chose to sort the nearest beacons array, so that the largest beacon was always at the end, avoiding having to iterate through the nearest beacons array for each beacon. This gained efficiency for smaller numbers of beacons to return, but lost it with larger numbers.

Testing

I have provided a class called FunctionalityTests to show that each of my algorithms functions as expected.

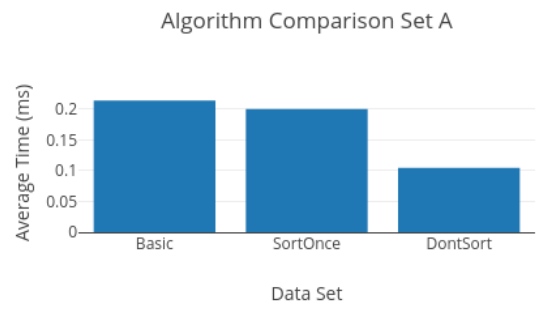
Testing that my methods to find the mean and standard deviation are accurate

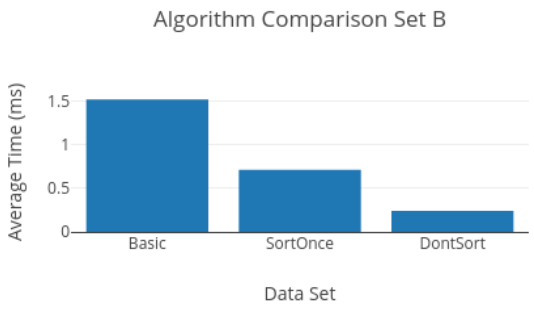
This is an example produced by running the Basic algorithm three times, to return the 3 nearest beacons.

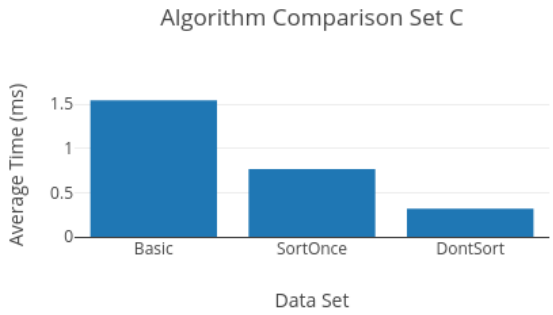
```
Distances:
6.4031242374328485
10.295630140987
12.206555615733702
Time taken: 0.685495 milliseconds

Distances:
12.36931687685298
13.341664064126334
16.64331697709324
Time taken: 0.548322 milliseconds

Distances:
5.0
7.0
16.55294535724685
Time taken: 0.490329 milliseconds

Average time for Basic algorithm: 0.5747153333 milliseconds
Standard deviation: 0.0818327478

Process finished with exit code 0
```

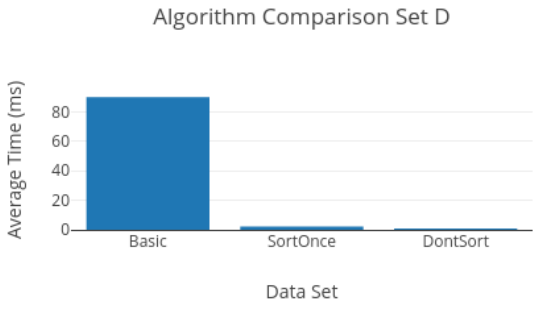
Times taken: 0.685495, 0.548322, 0.490329
Average time (mean): (0.685495 + 0.548322 + 0.490329)/3
= 1.724146/3
= 0.574715333
Squared differences:
(0.685495 – 0.574715333)^2 + (0.548322 – 0.574715333)^2 + (0.490329 – 0.574715333)^2
= (0.110779667^2) + (-0.026393333^2) + (-0.084386333^2)
= 0.01227213462 + 0.00069660802 + 0.00712105319
= 0.02008979583
Variance: 0.02008979583/3 = 0.00669659861
Standard deviation = √variance = √0.00669659861 = 0.08183274778 = 0.0818327478 to 10 dp.

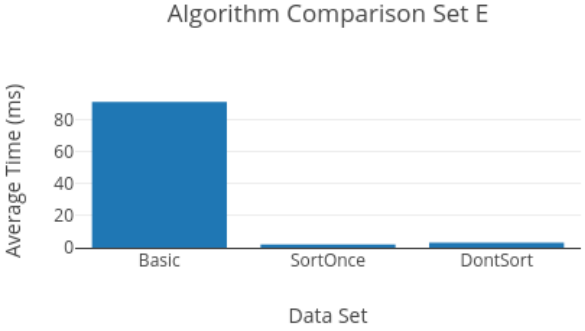
Algorithm comparison:
This group of bar graphs shows the performance of the various algorithms at given sets of values. I ran 100 tests for each algorithm and each set.
Set A: maxCoordinate = 10, totalBeaconNo = 100, nearestBeaconNo = 10
Set B: maxCoordinate = 100, totalBeaconNo = 1000, nearestBeaconNo = 10
Set C: maxCoordinate = 100, totalBeaconNo = 1000, nearestBeaconNo = 100
Set D: maxCoordinate = 1000, totalBeaconNo = 10000, nearestBeaconNo = 100
Set E: maxCoordinate = 1000, totalBeaconNo = 10000, nearestBeaconNo = 1000







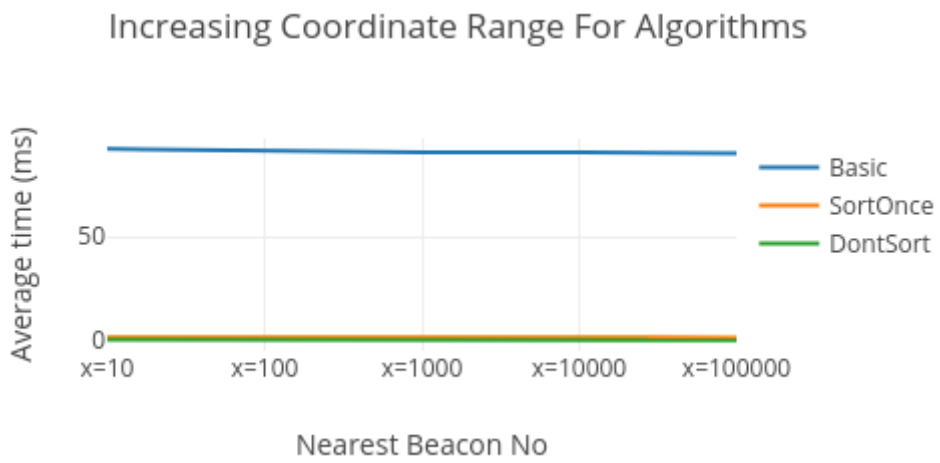




The algorithms take a similar amount of time on very small data sets, but the basic algorithm very quickly becomes very much more inefficient than the other algorithms, with a complexity of O(n^2). The DontSort algorithm is generally faster than the SortOnce algorithm, but sets D and E show that returning large numbers of the nearest beacons can make the DontSort algorithm slower, which is shown more clearly later.

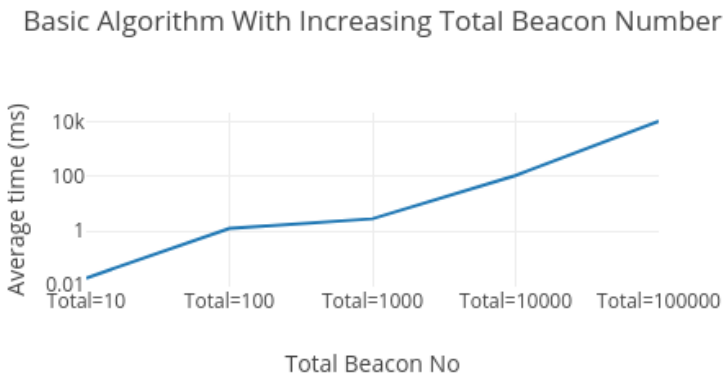
Unless otherwise stated in the following sets of tests, the values for minimum and maximum coordinates, total and nearest beacon numbers are the values in Main.java. I also include a dummy process at the start of the test to avoid the timings and standard deviation being affected by start-up processes.

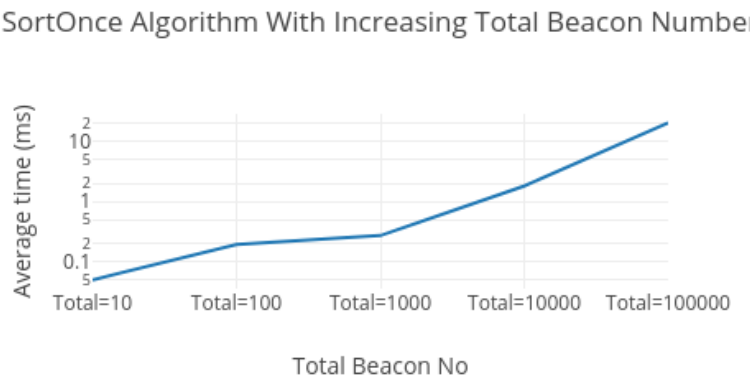
Testing how increasing the range of coordinates available for the beacons and target to be placed in affects the performance of the algorithms, by increasing the maximum coordinate:

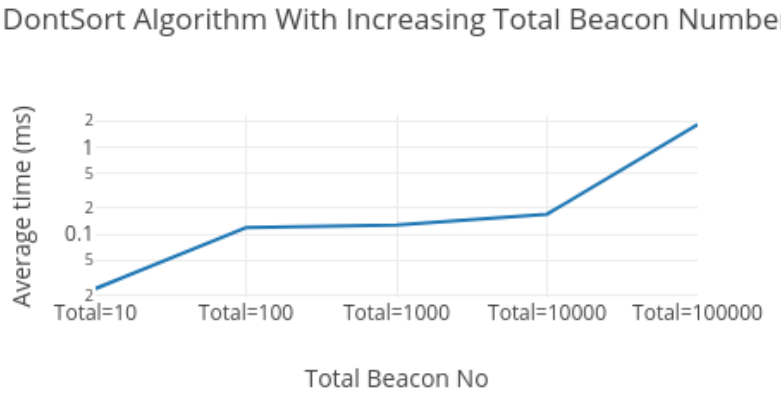


The straightness of the lines appears slightly exaggerated by zooming out to include the Basic algorithm in the graph, but it's clear that the range of coordinates available for the beacon and target to be placed in doesn't affect the time taken for the algorithms. This suggests that the Java functions for comparing and manipulating doubles are not noticeably affected by the size of the double.

Testing how increasing the total number of beacons affects the performance of the algorithms:



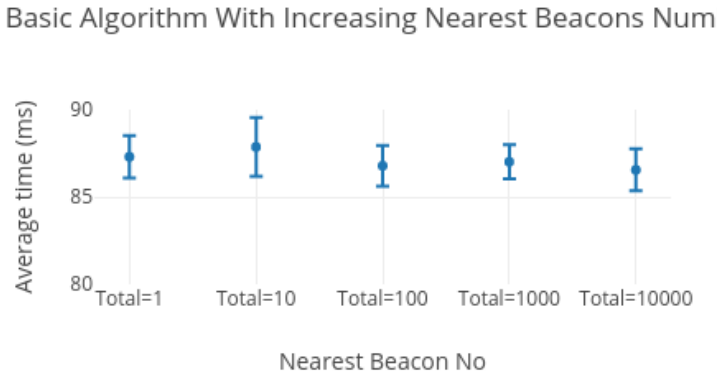




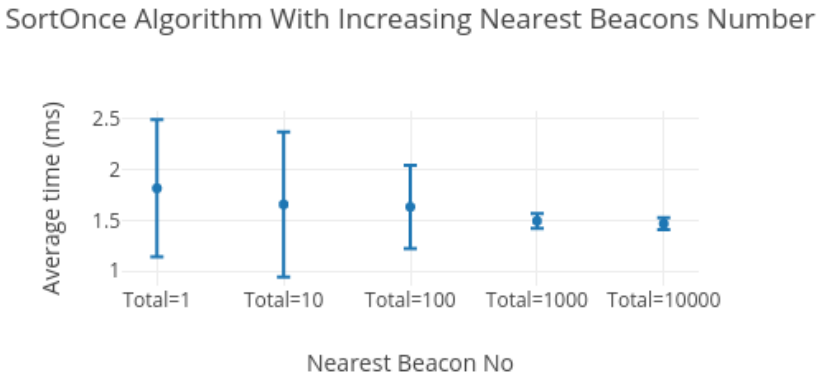
As the total number of beacons increases, the increase in time taken by each algorithm appears to increase in a similar way, when shown using logarithmic y axes. Despite the similarity of shape however, each algorithm's time complexity does increase at a different rate, as shown by the scales on the y axis.

Testing how increasing the number of nearest beacons returned affects the performance of the algorithms:

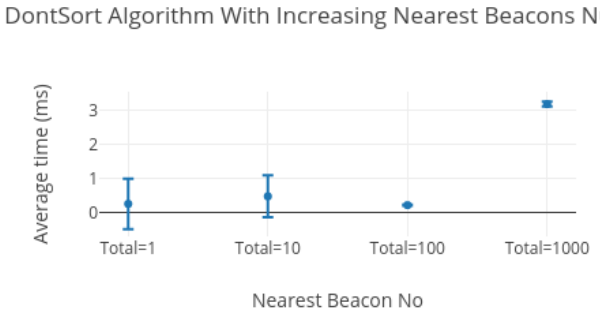
All of the following tests were run with the total beacon number set to 10000.

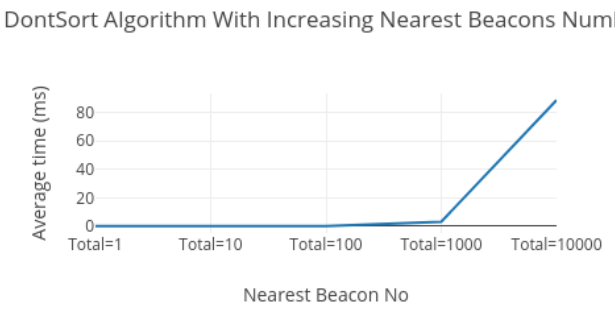


The number of nearest beacons makes no impact on the Basic algorithm, as expected as the algorithm sorts all the beacons anyway. The only computational requirement increased is the copying into the nearestBeacons array, which appears to have a minimal effect on efficiency.



When there is a smaller number of beacons to be returned, the standard deviation of the SortOnce algorithm is noticably large- this remained constant through several repetitions of the test, although I can't particularly think why that would be. Similarly to the Basic algorithm, the SortOnce algorithm sorts all the beacons anyway, so changing the number of nearest beacons doesn't affect time taken.

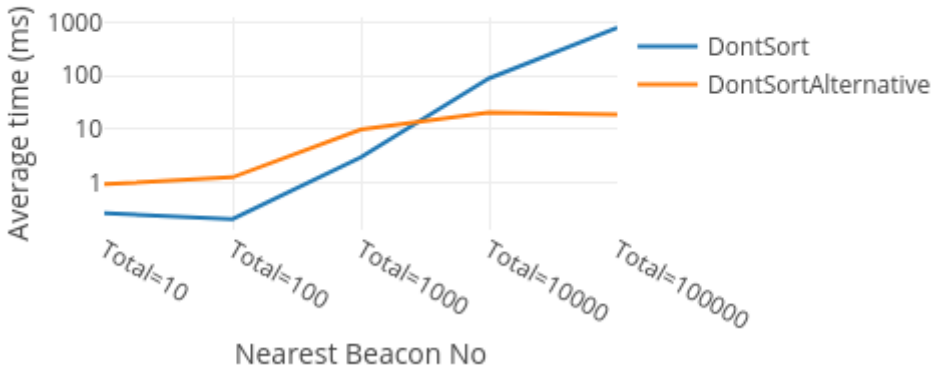




I've included two graphs for the DontSort algorithm, as its hard to make out any of the first points when the last is included. It's clear that as the number of nearest beacons increases, the DontSort algorithm becomes much less efficient. I think this is largely due to my inclusion of a insert sort in the nearest beacons list, which makes the algorithm more efficient than searching through the whole nearest beacons list to find the current largest for each beacon, but loses efficiency as the size of the nearest beacons list grows.

Extension

As the my implementation of the DontSort algorithm is inefficient at higher numbers of nearest beacons, I experimented with a different version of the algorithm. This is commented and explained in the extensionAlgorithms package. The graph below shows the efficiencies of the two algorithms as values of nearest beacons to be returned increase, with a logarithmic y axis scale.



Conclusion

I have successfully implemented and tested all of the sorting algorithms, describing and explaining the choices I made in implementation. I have presented the results of my experiments graphically above, and have analysed the results.